qTLB: Looking Inside the Look-Aside Buffer

Omesh Tickoo[1], Hari Kannan[2], Vineet Chadha[3], Ramesh Illikkal[1], Ravi Iyer[1], and Donald Newell[1]

[1] Intel Corporation, 2111 NE 25th Ave., Hillsboro OR, USA,
[2] Stanford University, Stanford CA, USA,
[3] University of Florida, Gainesville FL, USA

Abstract. Rapid evolution of multi-core platforms is putting additional stress on shared processor resources like TLB. TLBs have mostly been private resources for the application running on the core, due to the constant flushing of entries on context switches. Recent technologies like virtualization enable independent execution of software domains leading to performance issues because of interesting dynamics at the shared hardware resources. The advent of TLB tagging with application and VM identifiers, however, increases the lifespan of these resources. In this paper, we demonstrate that TLB tagging and refraining from flushing the hypervisor TLB entries during a VM context switch can lead to considerable performance benefits. We show that it is possible to improve the TLB performance of an important application by protecting its TLB entries from the interference of other low priority VMs/applications and providing differentiated service. We present our QoS architecture framework for TLB (qTLB) and show its benefits.

1 Introduction

CMP architectures are increasingly used for server and workload consolidation [8][9]. Industry trend is moving towards sharing the on-die and off-die platform resources across multiple heterogeneous applications or VMs running simultaneously on multiple cores of CMP systems. The success of CMP platforms depends not only on the number of cores but also heavily on the other platform resources (cache, memory, etc) available and their efficient usage. Traditionally, processor and platform architectures have been designed to perform well while running a single application. However, with the evolving software use models, CMP platforms are being geared towards running multiple applications simultaneously. The rapid deployment of virtualization [11][6] as a means to consolidate multiple applications on a platform is a prime example. When these disparate applications run simultaneously on CMP architectures, the quality of service (QoS) that the platform provides to each individual application will be non-deterministic (or chaotic) because it depends heavily on the behavior of the other simultaneously running workloads. As expected, recent studies [11][16][6] have indicated that contention for critical platform resources (e.g. cache, memory, I/O) is the primary cause for this lack of determinism. In this work we focus on the impact of virtualization on another major processor resource: translation look-aside buffer (TLB).

S. Aluru et al. (Eds.): HiPC 2007, LNCS 4873, pp. 107–118, 2007.

In order to design efficient virtualized systems on a CMP platform, the key challenge is to understand how micro-architectural features impact the performance of workloads in such environments. Recent studies [3] [2] show that significant performance overhead can be attributed to increased cache and TLB misses. TLBs are used to reduce the overhead of address translations in paging systems such as the x86. The TLB semantics mandate almost complete TLB flushes after context switches, in order to maintain consistency. While previous studies have relied on measurements to assess the performance impact of virtualization of existing workloads and systems, it is important to understand the impact of this new use model in the context of upcoming processor features like TLB tagging.

Typically, in a virtualized environment, process switches between different virtual machines (VMs) lead to complete TLB flushes. In typical consolidation environments, VM switching is often a very frequent event. Even though VMs in a virtualized environment are often scheduled based on different schedulers (such as BVT, SEDF in Xen) [11], the fun5damental problem of performance degradation due to TLB flushing on a context switch remains the same due to uncontrolled assignment and removal of TLB resources for applications running in different virtual machines. In fact, the TLB flushing behavior during frequent VM switches mitigates the advantage of faster address translations [2]. Our experimental results with SPEC CPU 2000 benchmarks support this argument. In the past, TLBs have been tagged with a global bit to prevent flushing of global pages such as shared libraries and kernel data structures. In some of the current system architectures, context switch overhead can be reduced by tagging TLB entries with address-space identifiers (ASID). A tag based on the virtual machine's ID (VMID) could be further used to improve I/O performance for virtual machines. New processor architectures, with hardware virtualization support, incorporate features such as virtual-processor identifiers (VPID) to tag entries in the TLB[4][10]. This level of tagging increases the longevity of TLB entries, and mitigates the performance penalty currently incurred on context switches.

Recent studies on shared resource management have either advocated the need for fair distribution between threads and applications, or unfair distribution with the purpose of improving overall system performance. The work presented here aims to extend these concepts to TLBs with a goal of improving the performance of an individual application at the cost of the potential detriment of others, with guidance from the operating environment. This is motivated by usage models such as server consolidation where service level agreements motivate the degree of performance differentiation [15] desired for some applications. Since the relative importance of the deployed applications is best managed by the operating software environment, we experiment with software-guided priorities (e.g. assigned by server administrators) to efficiently manage hardware resources. We compare the use of software-guided priorities (qTLB - QoS-aware TLBs) against non QoS-aware schemes. We also present the effect of scaling the TLB sizes (instruction and data), on application performance. Our full system simulation infrastructure is supplemented with detailed performance models for the caches and TLBs with QoS tuning knobs to be used by the system soft-ware. To our knowledge, this is the first study using full-system simulation to evaluate quality of service for TLBs using virtualized workloads for a CMP platform.

2 Analysis Methodology

In this section, we present an overview of our full system simulation analysis methodology. We choose to employ Xen VMM for workload characterization because it is a de-facto para-virtualized (split I/O) open source VMM. In our test framework we ported Xen VMM to run on a full system simulation environment. To identify the hardware TLB entries belonging to different VMs we needed to pass the VM information for each specific memory access to the hardware modules. We accomplished this by modifying the Xen hypervisor to provide this information on each context switch.

The Xen virtualized environment includes the Xen hypervisor, the service domain (Dom0) with its O/S kernel and applications, and a guest, "user" domain (DomU) with its O/S kernel and applications (Figure 1). This environment allows us to characterize different applications for workload characterization. The DomU guest uses a front end driver to communicate with a backend driver inside Dom0, which controls the I/O devices.

To evaluate the TLB dynamics in virtualized environments, we need an experimental framework that allows us visibility into both the hardware system architecture and the software stack. We chose SoftSDV[17] simulator for our studies. In the past, SoftSDV has been deployed to measure hardware resource usage under virtualized execution environments [19]. The simulation setup is shown in Figure 1. The execution-driven simulation environment combines functional and performance models of the platform. For this study, we chose to abstract the processor performance model and focus on detailed TLB models with QoS support to enable the coverage of multiple phases and a long execution period of the workload.

Figure 2 summarizes the profiling methodology and the tools we used. The following sections describe the individual steps in detail; these include (1) Full system simulation with virtualized workload, and (2) Performance simulation with QoS

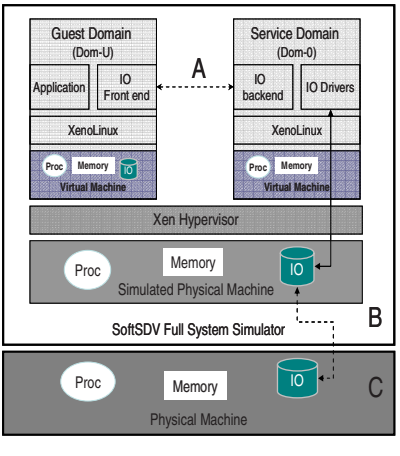


Fig. 1. Full system simulation environment Includes: (A) Xen Virtual Environment (B) SoftSDV Simulator (C) Physical Machine

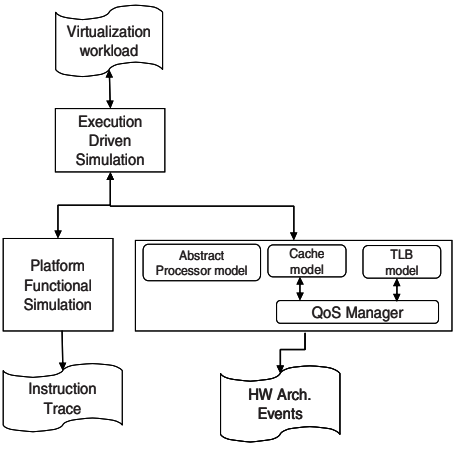


Fig. 2. Execution Driven Simulation profiling methodology: virtualization workload, functional simulation mode & performance simulation mode

services. The SoftSDV simulation framework was extended to support TLB QoS [19]. The simulation environment provides us with the capability of changing the underlying hardware architecture to evaluate architecture enhancements and their impact on workload performance. We tagged the TLB entries with their corresponding VMIDs, and added TLB QoS by enhancing the replacement algorithm. Our environment supported monitoring of TLB resources per VM, enforcement of QoS at the TLB level and an interface for software to communicate information about the currently running VM and individual VM priorities.

We calculated TLB utilizations while concurrently running applications in a virtualized environment. Figure 2 shows the QoS management module used to communicate with the abstract TLB model to provide QoS services. We employed a simple LRU based TLB replacement policy to evaluate the performance of various applications. QoS analysis is performed by using application level priorities to determine the percentage of TLB flushing and reservations. In addition, we considered the locality and working sets of the benchmarks for evaluation of our prototype. Section 3 discusses our proposed architecture in detail.

3 QOS-Aware Architecture

We propose a layered QoS architecture that implements static and dynamic QoS policies. Our proposed QoS-aware TLB architecture consists of three primary layers: priority enforcement, pri-ority assignment and priority classification.

The priority classification layer is responsible for identifying and providing e QoS information i.e. priority levels of each running application (e.g. 0 for high and 1 for low) and the associated targets/constraints. As shown in Figure 3, this layer requires support in the execution environment (either OS or hypervisor) as well as the processor architecture. Operationally, support (in the form of a QoS API) is required for the user or administrator to supply the required QoS in-formation to the execution environment. The support in the execution environment is the ability to maintain the QoS information in the thread state and the ability to save and restore it in the process architectural state when the thread is scheduled to run. The support in the processor is essentially a new control register called Platform QoS Register (PQR) needed to maintain the QoS information for the run time. The execution environment sets the PQR with the platform priority level of the currently running application. When static QoS assignments are used, different priority levels can be directly mapped to various resource utilization thresholds. In contrast, maintaining a pre-defined target performance level of an application during run-time entails the need for dynamic QoS strategies. The PQR register will be used to convey the mapping of priority levels into resource thresholds (for static QoS) and the mapping of priority levels to targets/constraints (in case of dynamic QoS). For priority assignment, resource targets are used for QoS level mapping, and to indicate the TLB occupancy thresholds for each priority level. In this paper, we only used static QoS policies.

Figure 3 illustrates the priority enforcement layer in the architecture and shows the components involved. The inputs to the enforcement layer are the tagged memory accesses and the QoS re-source table. As shown in Figure 4, each line in the TLB is tagged with a priority level in order to keep track of the current TLB space utilization

per priority level. The QoS resource table uses this information to store the TLB utilization per priority level. This is done simply by incrementing the resource usage when a new line is allocated in the TLB and decrementing the resource usage when a replacement or eviction occurs.

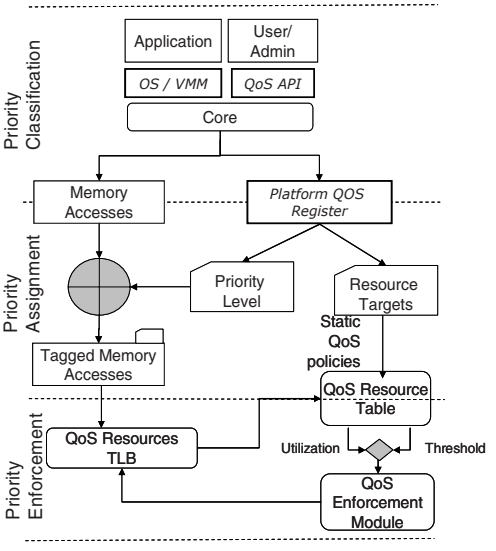


Fig. 3. QoS Architecture for TLB Resources

The static QoS policy is implemented by modifying the TLB replacement policy to be QoS aware. For each priority level, the utilization and the static QoS thresholds are available (in the QoS Resource Table - QRT) on a per priority level basis. If utilization of a priority level is lower than the specified threshold, then the replacement policy works in normal mode using the base policy (like LRU). When the utilization reaches the static QoS threshold, the QoS based replacement policy overrides the LRU policy to ensure that a victim is found within the same priority level (thus keeping the utilization for the priority level constant).

4 Experiments and Results

The goal of our experiments is to study the impact of various TLB configurations on virtualized workload performance. Keeping in mind that different applications have different working set sizes, our experiments are designed to evaluate the interaction between different applications at the TLB level. We also investigate the effect on individual applications due to different TLB QoS management policies in virtualized systems.

In following sections, we present the data TLB (DTLB) and instruction TLB (ITLB) study results. It is important to evaluate the performance of ITLB and DTLB separately because TLB access dynamics vary for data and instructions. We will first focus on the DTLB.

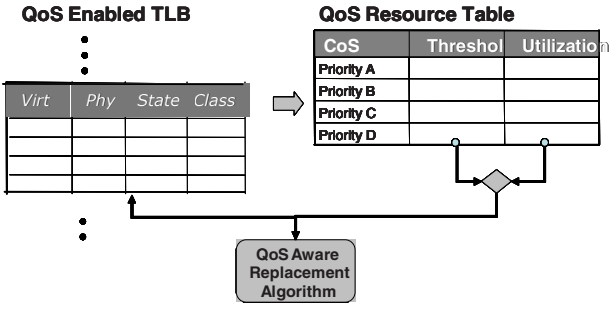


Fig. 4. QoS-Enabled TLB Enforcement

4.1 Data TLB

We evaluated DTLB performance of different workloads comprising the *CPU 2000* benchmark suite. The results show that in virtualized environments, per-application TLB resource requirements vary both with the application under consideration, and the set of other applications running concurrently. In scenarios of applications running concurrently on different VMs, different QoS mechanisms are needed to achieve the desired performance for high priority applications.

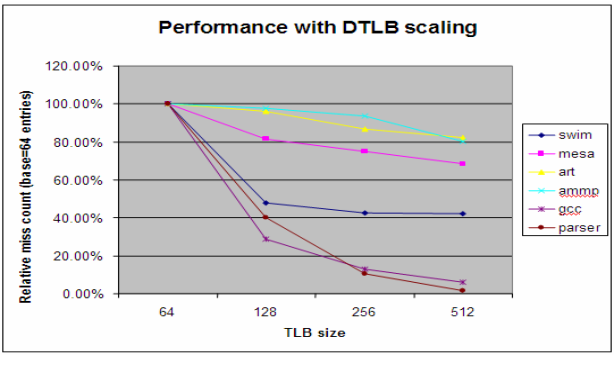


Fig. 5. Relative Change in data TLB miss rate with changing TLB size

To understand the TLB requirements for different applications, we broadly categorize the workloads on the basis of their working set sizes (Figure 5). Depending on the working set profile and data access locality, applications can fall into TLB friendly or TLB un-friendly categories. For description purposes we will categorize them into highly friendly, medium friendly and minimal friendly.

Highly TLB Friendly: These applications are characterized by a high degree of temporal and spatial locality. The applications tend to benefit highly with increased TLB resources. The more you give the better the performance is. Of the *SPEC CPU 2000* workloads we studied *parser* and *gcc* show this behavior of TLB locality.

Medium TLB friendly: The applications in this category show localization over a long range of memory addresses. Thus, while performance gains are evident at lower TLB sizes, increasing the TLB sizes leads to proportional decrease of the TLB miss count. Figure 5 illustrates this behavior for the *swim* benchmark.

Minimal TLB friendly: The working set for these applications exhibits a high degree of randomness in terms of addresses accessed. Therefore, TLB scaling has very little or no impact on the performance of such applications. From the *SPEC CPU 2000* suite, *ammp* and *art* (Figure 5) exhibit such behavior.

Next we will look into simultaneous execution of workloads and the impact of VMID tagging.

4.2 Impact of VM Tagging (VMID)

In current virtualization environments, a context switch from one VM to another leads to a complete TLB flush and subsequent repopulation of the TLBs from a clean state. Major processor manufacturers are employing TLB tagging with VMIDs in their new processor offerings. Tagging the TLB entries with global VMIDs and subsequently avoiding the flushing of these entries on a VM context switch will potentially improve the TLB performance considerably. The results from our experiments with VMID tagging are shown in Figure 6. We observe that depending on the nature of applications, significant reductions in DTLB miss count can be obtained by tagging the TLB lines with VMIDs which prevents flushing of the hypervisor mappings on context switches. Note that the graphs in Figure 6 show the percentage change in the miss count and not the absolute values of the miss count. In terms of absolute values, the miss count (or misses per instruction - MPI) of different workloads vary widely from each other depending on the nature of the individual workloads. But it can be observed that at small number of TLB entries, the impact of VMID is not that significant. As we increase the number of TLB entries, combinations with lower DTLB utilization benefit from tagging. This is due to the fact that a destructive application running after the context switch wipes the TLB out before the VM is scheduled again. One solution to this problem is to reduce the interference from the destructive VM through QoS as shown in the next section.

4.3 Impact of DTLB QoS

Our next step is to understand the TLB level interactions between multiple applications with different working set sizes and the effect of TLB QoS on performance. In our simulation setup, two different applications are run under the Xen virtualization environment with one workload running in Domain-0 and another one in a dedicated virtual machine. The TLB footprint obtained for Domain-0 is influenced by the combination of the test workload running in Domain-0 and other Xen related processes running in the administrative domain. To understand the effect of TLB QoS on an individual workload performance, we assign higher priority to the workload running in the isolated VM. The exact QoS metrics are tunable and are described in detail below. We use VMIDs to tag the TLB entries for QoS enforcement

purposes. In the graphs presented below, we plot the miss ratios for high priority applications with changing occupancy limits for the lower priority application. The miss counts are plotted relative to the miss count when no QoS is enforced. Table 1 shows the different TLB configurations analyzed.

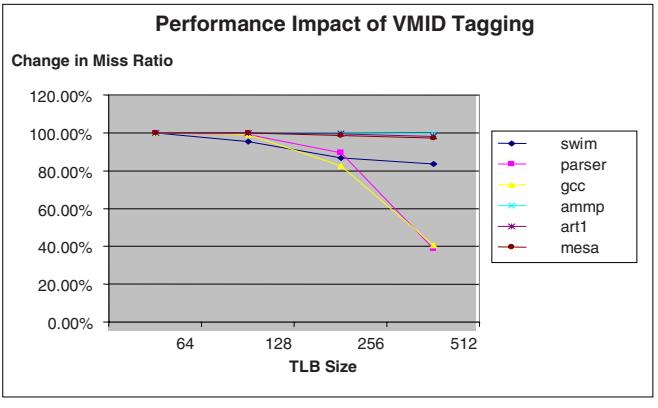


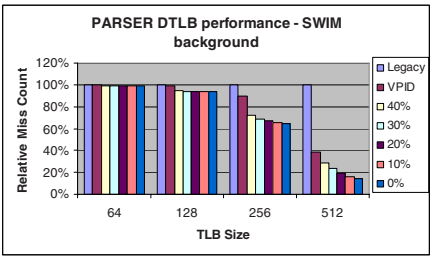
Fig. 6. DTLB performance impact of VMID tagging

Consider a scenario where the high priority application exhibits characteristics of highly TLB friendly workload. Since, the applications benefit from being allocated more entries in the TLB, restricting the background app will provide considerable performance improvement. This is more significant when the TLB size is small. Better management of the TLB can provide better results for the important application in this scenario. We will look at two sets of results to demonstrate this behavior. The first set of results (Figure 7) uses SWIM as the background process. It may be noted that the highly TLB friendly applications *gcc* and *parser* benefit highly from the increased TLB resources provided by TLB QoS. On the other hand, *art* and *ammp* which are minimal TLB friendly get minimal benefit out of TLB QoS.

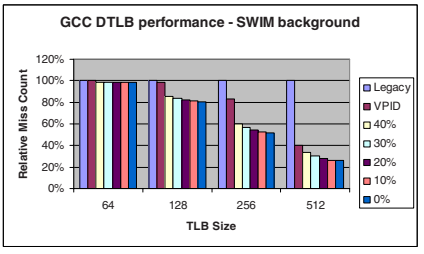
Another important observation is that the VMID tagging benefits dwarfed by the excessive TLB resource utilization are now moderated by employing TLB QoS.

Table 1. TLB configurations supported

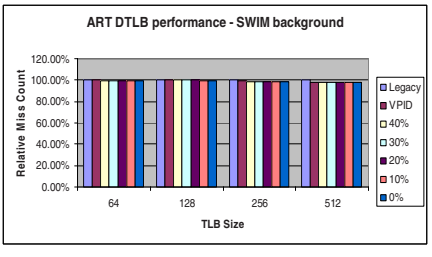
System Scenarios	TLB Semantics
Legacy System	TLBs are flushed on each context switch.
VMID tagging (No application TLB QoS)	VMID tagging and TLB entries are not flushed on VM switches. LRU is used to replace the TLB entries across VMs.
X% (preferential Resource allocation)	VMID tagging with QoS Aware replacement. Low priority application gets at most X% of the TLB capacity. X= 40, 30, 20, 10, 0 (examples)



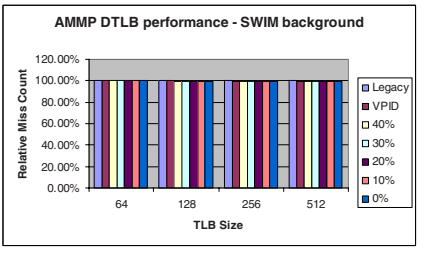
(a) Miss Rates for *parser* in *swim* vs. *parser* (*parser* has higher priority)



(b) Miss Rates for *gcc* in *swim* vs. *gcc* (*gcc* has higher priority)



(c) Miss Rates for *art* in *swim* vs. *art* (*art* has higher priority)



(d) Miss Rates for *ammp* in *swim* vs. *ammp* (*ammp* has higher priority)

Fig. 7. Impact of VMID and TLB QoS on various applications with SWIM in background

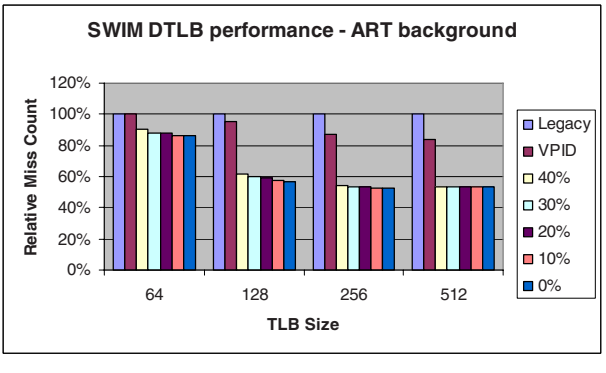


Fig. 8. Miss Rates for *swim* in *art* vs. *swim* (*swim* has higher priority)

It should be noted that the impact of TLB QoS depends both on the foreground as well as on the background application. Results with *art* as a background application are shown in Figure 8. Since *art* is less TLB intensive than *swim*, the impact of *art* on the foreground application is considerably less. This results in better QoS results even with smaller TLB sizes. The *art* vs *swim* plot shows that TLB QoS is needed to ensure that the a minimum number of entries must necessarily stay dedicated for the high

priority application. The performance gains for *swim* reduce after a minimum TLB size of 128 entries is reached or when the TLB QoS mechanism ensures a minimum level of allocation for *swim*.

4.4 ITLB QoS

Locality behavior of instructions is different than that of data. Applications typically have small code working sets that fit into smaller TLBs. They also exhibit a high degree of locality. Instruction TLB behavior with TLB scaling is shown in Figure 9. We note that with increase in the size of the TLB, relative miss ratio decreases and is almost constant after size of 128. We infer that an ITLB size of 128 entries is sufficient to incorporate almost all possible address translations during the TLB stage, hence reducing the performance penalty.

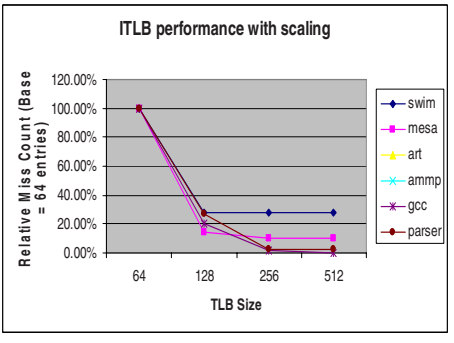


Fig. 9. ITLB Scaling Impact

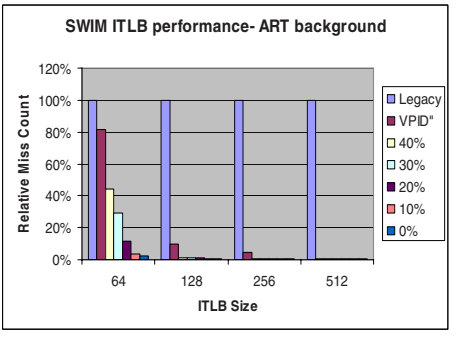


Fig. 10. ITLB Miss Rates for *swim* in *art* vs. *swim* (*swim* has higher priority)

Intuitively, we expect that most applications will have a fairly smaller instruction working set when compared with the data working set. Our experimental results support this intuition. Consequently, to improve the ITLB hit rates for higher priority applications in heterogeneous operating environments, we can either increase the size of ITLBs to a minimum acceptable level (128 entries for high priority VM from Figure 9), or tune the QoS factor to achieve equivalent capacity for the high priority application. It should also be noted that VMID tagging alone works well with all TLB sizes.

As Figure 10 shows, high gains are obtained at moderate ITLB sizes and moderate capacity restrictions for low priority applications. In fact, ITLB size of 128 entries and a QoS factor close to 0.5 ensuring fair distribution of the ITLB provides close to maximum performance boost. QoS tuning beyond this point does not produce proportionate results.

This type of behavior was observed in all the studied workloads leading us to conclude that providing ITLB QoS in virtualized systems is less application sensitive than the DTLB QoS and may amount to ensuring a fair TLB distribution in most cases.

5 Conclusion and Future Work

Virtualization and multi-core architectures are two complementary upcoming paradigms that throw open interesting workloads and applications scenarios. In this paper we analyzed the TLB level interactions of different applications operating in virtualized settings. Our execution driven simulation based results show that modifications to default TLB management policies are needed for efficient operation in such settings. We show that using VMIDs to avoid flushing the global (VMM) entries from TLBs on VM context switches leads to significant drops in TLB miss rates.

We also investigated the effect of prioritizing the applications and providing QoS in terms of TLB capacity. Our investigations show that different applications display different TLB related behaviors depending on the working set sizes and access locality. Running multiple applications within different virtual machines raises interesting TLB sharing scenarios. In such conditions, our experiments show that an administrator can potentially provide a preferential performance boost to high priority applications using TLB QoS. The knowledge of application working-set sizes and access locality can be used to determine the QoS factors needed for a targeted TLB miss count.

We are investigating how QoS services will affect TLB coherence protocols in context of performance and overhead. We are currently in the process of designing a dynamic TLB QoS policy that tunes the QoS factor during run-time to achieve a guaranteed minimum performance level for high priority applications. We are also investigating hardware and software enhancements for architecting QoS aware multi-core platforms.

References

[1] Foong, A., Fung, J., Newell, D.: An In-Depth Analysis of the Impact of Processor Affinity on Network Performance. In: Proceeding of IEEE Int'l Conf. Networks, IEEE Press, Los Alamitos (2004)

[2] Menon, A., Cox, A., Zwaenepoel, W.: Optimizing Network Virtualization in Xen, USENIX Annual Technical Conference (2006)

[3] Chandra, D., Guo, F., Kim, S., Solihin, Y.: Predicting inter-thread cache contention on a chip multiprocessor architecture. In: HPCA. Proc. 11th International Symposium on High Performance Computer Architecture (February 2005)

[4] Neiger, G., Santoni, A., Leung, F., Rodgers, D., Uhlig, R.: Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. Intel Technology Journal (August 2006)

[5] Kannan, H., Guo, F., Zhao, L., Illikkal, R., Iyer, R., Newell, D., Solihin, Y., Kozyrakis, C.: From Chaos to QoS: Case Studies in CMP Resource Management. In: dasCMP/ MICRO. From Chaos to QoS: Case Studies in CMP Resource Management, 2nd Workshop on Design, Architecture and Simulation of CMP platforms (December 2006)

[6] Intel Virtualization. Technology Specification for the IA-32 Intel Architecture (April 2005)

[7] Hsu, L., Reinhardt, S., Iyer, R., Makineni, S.: Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches as a Shared Resource. In: 15th International Conference on Parallel Architectures and Compilation Techniques (PACT) (September 2006)
[8] Nesbit, K.J., et al.: Fair Queuing Memory Systems. MICRO (2006)
[9] Marty, M.R, Hill, M.D.: Virtual hierarchies to support server consolidation. In: proceedings of ISCA 2007 (2007)
[10] Pacifica, – Next Generation Architecture for Efficient Virtual Machines (Accessed, April 2007), http://developer.amd.com/assets/WinHEC 2005_Pacifica_Virtualization.pdf
[11] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the Art of Virtualization. In: Proceedings of the ACM symposium on operating systems principles (October 2003)
[12] Illikkal, R., Iyer, R., Newell, D.: Micro-Architectural Anatomy of a Commercial TCP/IP Stack. In: WWC-7. 7th IEEE Annual Workshop on Workload Characterization (October 2004)
[13] Iyer, R.: CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms. In: ICS 2004. 18th Annual International Conference on Supercomputing (July 2004)
[14] Iyer, R.: On Modeling and Analyzing Cache Hierarchies using CASPER. In: Calzarossa, M.C., Gelenbe, E. (eds.) MASCOTS 2003. LNCS, vol. 2965, Springer, Heidelberg (2004)
[15] Iyer, R., Zhao, L., Guo, F., Illikkal, R., Makineni, S., Newell, D., Solihin, Y., Hsu, L., Reinhardt, S.: QoS Policies and Architecture for Cache/Memory in CMP Platforms. In: ACM SIGMETRICS 2007 (2007)
[16] Goldberg, R.P.: Survey of virtual machine research. IEEE Computer, 34–45 (1974)
[17] Uhlig, R., Fishtein, R., Gershon, O., Hirsh, I., Wang, H.: SoftSDV: A Presilicon Software Development Environment for the IA-64 Architecture. Intel Technology Journal. Q4 (1999), http://www.intel.com/ technology/itjf
[18] Makineni, S., Iyer, R.: Performance Characterization of TCP/IP Packet Processing in Commercial Server Workloads. In: 6th IEEE Workshop on Workload Characterization (October 2003)
[19] Chadha, V., Illikkal, R., Moses, J., Iyer, R., Newell, D., Figueiredo, R.J.: I/O Processing in a Virtualized Platform: A Simulation-Driven approach. In: Proceedings of VEE, San Diego (June 2007)